Dangerously Clever X1 Application Tricks

J.B. White III
Oak Ridge National Laboratory

Abstract

We describe optimization techniques on the Cray X1 that are either profoundly unportable or counterintuitive. For example, one can use small, static coarrays, Cray pointers, and the "`volatile`" attribute to pass arbitrary high-bandwidth, minimal-latency messages with no procedure-call overhead. Also, it may be advantageous to bring "`if`" statements inside "`do`" loops for vectorization. This paper describes how and why.

1 Introduction

We have come across a number of optimization techniques on the Cray X1 that are counterintuitive at first glance. The techniques described here are the following, with titles that appear to contradict other well-known techniques.

- Avoid using cache.
- Replace BLAS calls with "`do`" loops.
- Minimize vector length.
- Move "`if`" statements inside loops.
- Use more pointers.
- Add infinite loops.

In the following sections, we describe each optimization technique in greater detail and provide an example where it has improved performance. We end each section with a short assessment of the applicability of each technique to other applications. Because discussion of each optimization is self contained (the last two optimizations work together), we have included concluding remarks at the end of each section instead of collecting them in a final section.

2 Avoid using cache

The Cray X1 has the ability to load and store directly between memory and vector registers, without storing to the cache. This non-allocation of cache can improve performance when the memory operations have no temporal or spatial locality. An example with such a lack of locality is the strided "triad" benchmark, which has been used by Oliker *et al.* [1] in performance analyses. The benchmark is based on the "triad" benchmark from the STREAM suite [2], which measures the performance of element-wise vector multiplication and addition, written in Fortran array notation as follows.

```
a(:)=b(:)+s*c(:)
```

The strided triad benchmark measures the effect of strided memory access on performance. We have implemented this benchmark by timing the following operation on one X1 multi-streaming processor (MSP) for "`stride`" values of 1–500 with vectors of 10^8 double-precision real elements, where "`s`" is a double-precision real scalar.

```
a(::stride)=b(::stride)+s*c(::stride)
```

Fig. 1 compares the performance of our implementation with and without cache allocation. Avoiding allocation is implemented with a Cray directive that lists the variables that should not be cached. For our strided triad benchmark, the vector declarations and the directive take the following form.

```
real(8),allocatable::a(:),b(:),c(:)
!dir$ no_cache_alloc a,b,c
```

The results in Fig. 1 compare the performance with and without the second line above.

Outside of bank conflicts, the strided triad with caching achieves about 4 GB/s, while the non-cached triad achieves over 10 GB/s. This is a significant improvement.

Despite the magnitude of the improvement for this benchmark, this optimization is used rarely in applications. It is unusual for applications to have most

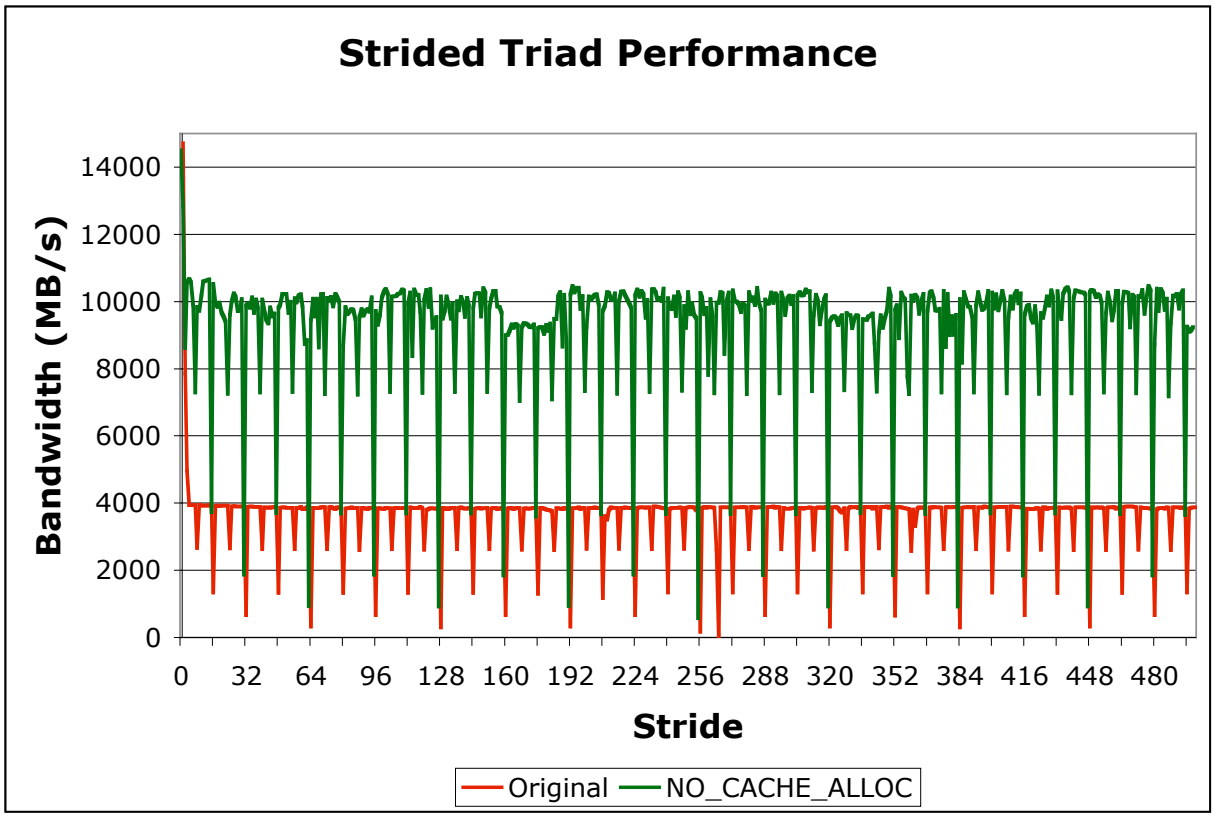

Figure 1: Performance of strided triad benchmark on one MSP with and without allocating cache.

memory operations with such an extreme lack of locality as in the benchmark.

3 Replace BLAS calls with "do" loops

Vendors of systems for high-performance computing (HPC) often provide highly optimized libraries for the Basic Linear Algebra Subprograms (BLAS) [3], and this is the case for Cray. Performance often improves dramatically when Fortran "do" loops or C "for" loops are replaced by the corresponding BLAS calls.

Early in the lifetime of the X1, some BLAS calls were not yet highly optimized, and the corresponding "do" loops could outperform them because the compiler could take advantage of the context to perform additional optimizations. We found this to be true for the BLAS3 call CGEMM, for example, where CGEMM performs matrix-matrix multiplication using single-precision complex matrices.

CGEMM and other calls have since been improved, reducing the need to replace them with "do" loops. Certain cases still exist, however, where the equivalent "do" loops perform better, typically in situations where the compiler can perform additional optimizations with inlined loops that it cannot with a subroutine call.

An example is a benchmark we used to measure the performance of DGER, a BLAS2 operation that performs a rank-one vector update of a double-precision real matrix. The benchmark performs a number of DGER operations on a matrix of size 4480×4480, as described in [4]. Because of the relatively small size of the test matrix, the benchmark performs many iterations to get accurate timing results. In its original form, the benchmark compared the times for the following two equivalent loops.

```
do iter=1,niters
   call dger(n,n,alpha,x,1,y,1,a,n)
end do

do iter=1,niters
   do j=1,n
      do i=1,n
         a(i,j)=a(i,j)+alpha*x(i)*y(j)
      end do
   end do
end do
```

With "n=4480" and "niters=100", performance for the DGER version on a single MSP was 2.4 GF, or 20% of the peak performance of 12.8 GF. In contrast, the timing results for the "do"-loop version indicated a performance of 138 GF, or 1078% of peak!

The Cray Fortran compiler can produce a listing of the source code, called a "loopmark", that identifies what optimizations were performed on each loop. The loopmark listing for the above loops reveals the source of the unbelievable performance.

```
Di--------< do iter=1,niters
Di Mr-----<    do j=1,n
Di Mr Vm--<       do i=1,n
Di Mr Vm             a(i,j)=
Di Mr Vm-->       end do
Di Mr----->    end do
Di--------> end do
```

The "Vm" indicates that the "i" loop was vectorized at that level ("V") and multistreamed at a higher level ("m"). The "Mr" indicates that the "j" loop was multistreamed ("M") and unrolled ("r"). But it is the "Di" that indicates the source of the performance; the "iter" loop—the timing loop—was interchanged with another loop ("i") and *deleted* ("D")!

The compiler was able to modify the computation to eliminate the timer loop. The three nested loops from above became the logical equivalent of the following.

```
nalpha=niters*alpha
do j=1,n
   do i=1,n
      a(i,j)=a(i,j)+nalpha*x(i)*y(j)
   end do
end do
```

The reported timings were thus based on many fewer operations than expected, so the resulting calculation of flops was incorrect.

The benchmark was modified to eliminate this optimization, for it was intended to represent an application where each DGER operation depended on the result of the previous one. Thus the optimization was not effective for the real application.

Other applications may include BLAS2 or BLAS1 calls within independent loops, in which case replacing them with explicit loops could allow "strength reduction" optimizations like the one above. We expect Cray to provide the ability to inline some BLAS1 calls in the near future, reducing the need to consider this optimization.

Replacing BLAS3 calls is unlikely to improve performance in any case because they already represent triply nested loops and are more amenable to specialized tuning.

4 Minimize vector length

On previous generations of Cray vector systems, long vectors achieve performance closer to peak than short vectors because they better amortize the latencies and startup costs of pipelined operations. Long vectors place significant requirements on memory bandwidth, however, but these previous vector systems have the bandwidth to meet these requirements for the computation rates they can sustain.

The same is not true for the X1. Though it has greater bandwidth than previous systems, its computation rate has increased even more. By splitting loops into blocks the size of the vector registers, one can improve performance by enhancing locality within the registers and cache and thus reducing the requirements for memory bandwidth. Also, the compiler can eliminate one level of the loop nest by replacing it with vector instructions.

We used the register-blocking technique to improve the performance of the FT benchmark from the NAS suite of parallel benchmarks [5]. FT performs distributed 3D FFTs that have a blocking size as input, and Fig. 2 compares performance of the FT class-C benchmark problem for blocks of size 64 and of the size that gives four blocks per MSP. Having four blocks per MSP maximizes the block size while allowing for multistreaming over blocks. The block size of 64 allows the inner looping to be eliminated in favor of vector instructions.

We used loopmarks to determine if the looping was eliminated; marks of "`Vs`" indicate vectorization with "short loops", loops short enough to be implemented as vector instructions with no actual looping. There were five cases where the compiler could not determine that a loop was "short", so we added the following directive to each.

```
!dir$ shortloop
```

Fig. 2 shows that the short-loop version consistently outperforms the four-block version over the range of MSP counts tested. The performance is plotted on a log scale, however, so it is difficult to gauge the magnitude of the performance increase. Fig. 3 shows the relative improvement of the short-loop version for each MSP count; it outperforms the maximum-block version by 1.6–1.8×.

The technique of blocking for vector registers is likely to be effective for many applications, though it may not have as great an effect for full applications as it does for small benchmarks like FT. It may be a "next step" for per-processor tuning, after vectorization and multistreaming, particular for systems like the Cray X1E, which will have substantially better floating-point performance than the X1 but with the same memory subsystem.

5 Move "`if`" statements inside loops

It is common practice to try to keep "`if`" statements outside of loops when the "`if`" test is independent of loop iteration; this avoids redundant tests and branches during execution. We have come across situations, however, when it is better to pull "`if`" statements back inside loops to allow multiple loops to be fused together. Fusing the loops may allow work arrays to be demoted to scalars, which the vectorizing compiler then promotes to vector registers, increasing register re-use and eliminating loads and stores to main memory.

One example is the "`state`" subroutine in the Parallel Ocean Program (POP) [6], version 1.4.3. This Fortran subroutine has a number of optional arguments, and it must check for the existence of those arguments before computing their values. Here is an excerpt from the unmodified "`state`", where the variables written in all capital letters are arrays.

```
 if (present(RHOOUT)) then
    RHOOUT=merge(((unt0+RHO0)*
$       BULK_MOD*DENOMK)*p001,
$       c0,KMT>=k)
 endif
 if (present(RHOFULL)) then
    RHOFULL=merge(((unt0+RHO0)*
$       BULK_MOD*DENOMK)*p001,
$       c0,KMT>=k)
 endif
```

The unmodified subroutine is written as a series of array statements, as illustrated above. The compiler can fuse array statements within the same code block, thus re-using registers and cache variables somewhat, but it does not fuse loops across "`if`" statements.

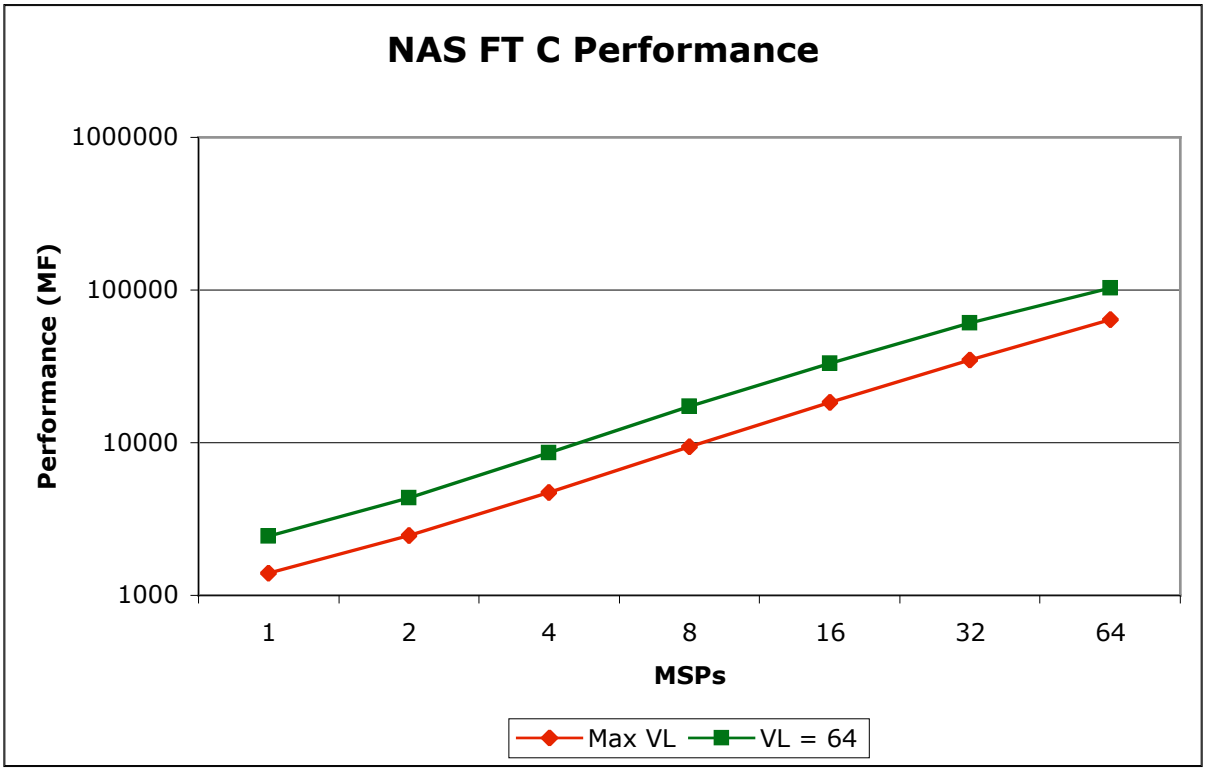

Figure 2: Performance of the NAS FT benchmark for the class C problem using various MSP counts, comparing a blocking factor of 64 ("VL=64") with blocking such that each MSP has four blocks ("Max VL").

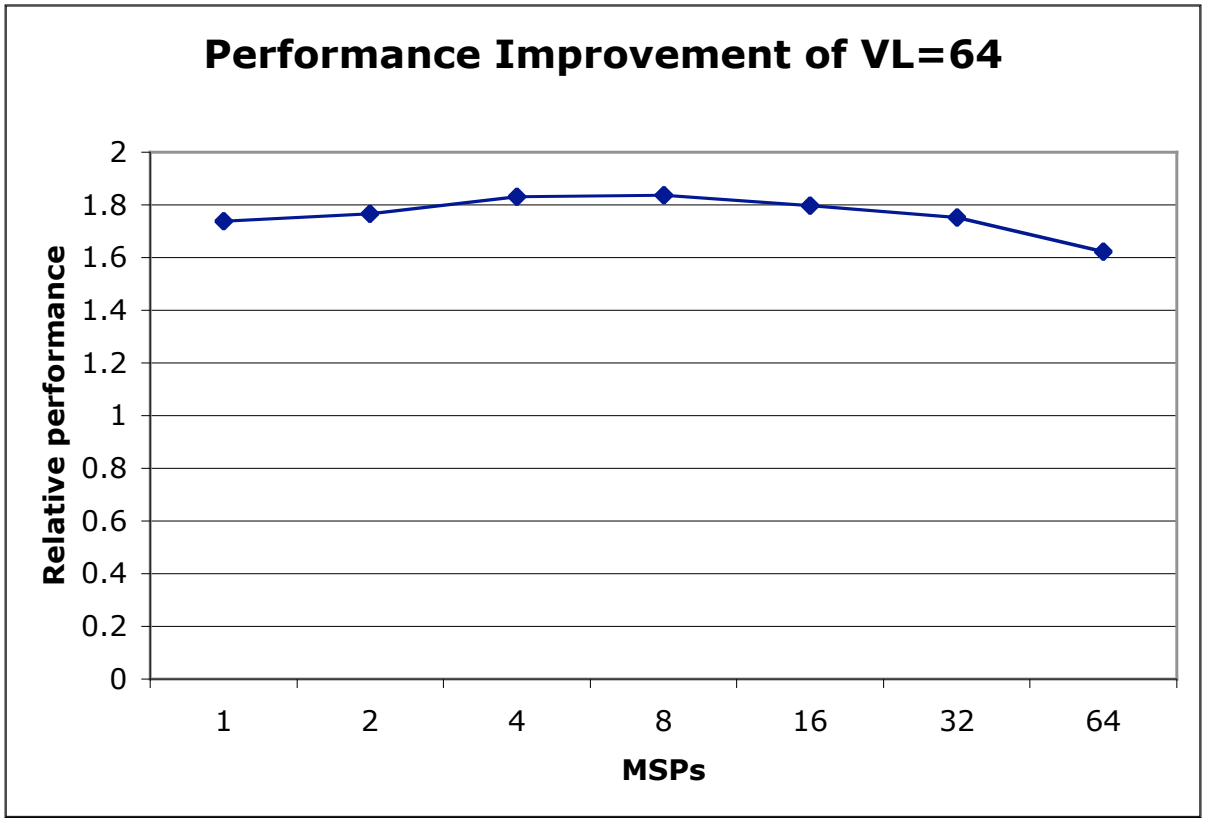

Figure 3: Relative improvement of the short-loop version of the FT benchmark over the original version with four blocks per MSP.

The entire subroutine can be written as a single loop nest by moving the "`if`" statements inside. This fusion of all the array statements allows many of the temporary arrays to be demoted to scalars. Here is the resulting code for the above excerpt. Note that many arrays (all upper case before) are now scalars (all lower case).

```
 do j=1,jmt ; do i=1,imt
    ...
    if (present(RHOOUT)) then
       RHOOUT(i,j)=merge(((unt0+rho0)*
$        bulk_mod*denomk)*p001,
$        c0,kmt_mask)
    endif
    if (present(RHOFULL)) then
       RHOFULL(i,j)=merge(((unt0+rho0)*
$        bulk_mod*denomk)*p001,
$        c0,kmt_mask)
    endif
    ...
 end do; end do
```

To measure the performance improvement, we did before and after runs of the POP benchmark problem with 1° resolution on one MSP. We instrumented each executable using "`pat_build`" and ran with the environment variable "`PAT_RT_EXPERIMENT`" set to "`samp_cs_time`". The results from "`pat_report`" are the following. Before (using array statements):

```
|16.3%|33.7%|19545|state@state_mod_
```

After (fused into one loop nest):

```
|13.5%|31.7%|15663|state@state_mod_
```

The "`state`" subroutine went from 16.3% of runtime to 13.5%, and from 19,545 profile samples to 15,663, an improvement of 25% for this subroutine.

Moving "`if`" statements inside loops is a specific technique for the general strategy of reducing memory-bandwidth requirements by blocking for registers and cache. After moving an "`if`" statements into a loop nest, it is important to confirm through loopmarks that the nest still vectorizes and multistreams. One should also consider the tradeoff between the performance improvement and any reduction in readability or maintainability of the resulting source code.

6 Use more pointers

HPC programmers should typically avoid the use of pointers because they limit or inhibit a compiler's ability to analyze dependences, thus limiting or inhibiting many optimizations. One reason to use pointers on the X1 is to improve performance of parallel communication; pointers can be used to replace MPI library calls [7] with direct loads and stores to remote memory.

Co-Array Fortran (CAF) [8] provides a way to create distributed arrays and access elements of such arrays remotely. A problem with CAF, however, is that its use in one procedure can force modification of all the procedures that call it, along with all the procedures that call those, and so on. This propagation of changes comes because an argument promoted to a co-array must be declared as a co-array in any procedure that might call the procedure.

Pointers can be combined with CAF to "cheat" on the X1 and take advantage of the fact that the X1 memory is globally addressable. We use the term "cheat" because one cannot assume global addressability for all implementations of CAF.

One can use the following steps to replace MPI calls with remote memory operations for arbitrary arguments to a procedure, without requiring any modifications to calling procedures.

- Declare a co-array of "`INTEGER(8)`".
- Declare a Cray pointer on the receiving process.
- The sender stores an array address in the receiver's co-array location, where the address comes from the non-standard intrinsic function "`loc`" applied to the sender's array.
- The receiver sets the pointer to the local value of the co-array, the value just assigned by sender.
- The receiver uses the pointer to access the sender's array.

An example of this technique is our modification of the "`global_scatter`" subroutine from CICE [9], a global model of ocean ice used for climate studies. Consider the following excerpt from the modified subroutine, which lists the code relevant to the master processor in the scatter operation.

```
 integer(8)::
$   remote_address(NPROC_X*NPROC_Y)[*]
 real(dbl_kind)::
$   workg(imt_global,jmt_global)

 integer(8)::address
```

```
 if (my_image==master_image) then
   address=loc(workg)
   do i=1,num_images()
     remote_address(master_image)[i]=
$       address
   end do
 end if

 ... ! Synchronize
```

First note that the co-array is also an array of size "NPROC_X*NPROC_Y", which is the total number of processors in the 2D decomposition used by CICE. One might think of the co-array as two dimensional, where one dimension is local to each process and the other is distributed over processes.

The master process takes the address of the global work array, "workg", and distributes that address to all processes by copying the address to the location indexed by the process number of the master.

The following excerpt then lists the code for the operation on all processes.

```
 integer(8)::
$   remote_address(NPROC_X*NPROC_Y)[*]
 real(dbl_kind)::work(ilo:ihi,jlo:jhi)

 real(dbl_kind) ::
$   workg_remote(imt_global,jmt_global)
 pointer(workg_address, workg_remote)

 ... ! Synchronize

 workg_address=remote_address(master_image)
 work(ilo:ihi,jlo:jhi)=
$   workg_remote(ilog:ihig,jlog:jhig)
```

After synchronizing, each process assigns the pointer "workg_address" to the address provided by the master. Each process knows to read the location indexed by the process number of the master. For the scatter operation, the co-array does not really need to be a local array also, but making it a local array allows the implementation of more-complex operations, such as all-to-all or many-to-many communication.

Once the pointer is assigned, each process can read directly from the remote array as if it were local.

In the excerpts above, we indicate where synchronization is necessary but not how it is implemented. The following section describes the implementation.

7 Add infinite loops

We implement synchronization for the above example using a "logical" co-array and the "integer(8)" co-array itself. These implementations include what appear to be infinite loops; they rely on external processes to change the value of their test expressions. The technique is as follows.

- All processes initialize the "integer(8)" co-array to zero.
- The sender initializes a "logical" co-array to false.
- The sender stores the address to the receivers' co-arrays, and the address is guaranteed to be nonzero.
- The recievers spin-wait for a nonzero address.
- After performing the necessary memory operations, each receiver sets to true the value of the "logical" co-array on the sender indexed by the receiver's process number.
- The sender spin-waits for true values from each receiver.

It is important that each process continues to read the values used in the loop tests from main memory, so the compiler must be instructed to not load the values into registers and simply reread the registers. The "volatile" attribute provides just this instruction. The following excerpt from "global_scatter" demonstrates the technique. Note that the co-arrays now have the "volatile" attribute.

```
 integer(8),volatile::
$   remote_address(NPROC_X*NPROC_Y)[*]
 logical, volatile::
$   remote_flag(NPROC_X*NPROC_Y)[*]
 ...
 do while (remote_address(master_image)==0)
 end do
 ... ! Copy data
 remote_address(master_image)=0
 remote_flag(my_image)[master_image]=.true.

 if (my_image==master_image) then
   do i=1,num_images()
     do while (.not. remote_flag(i))
     end do
     remote_flag(i)=.false.
   end do
 end if
```

To measure the performance improvement of the co-array/pointer implementation of "global_scatter", we compared production runs of a coupled POP/CICE model on eight MSPs for ten simulated days. We instrumented each executable with "pat_build" and ran with the environment variable "PAT_RT_EXPERIMENT" set to "samp_cs_time".

The original MPI implementation of "global_scatter", which uses "mpi_isend", "mpi_irecv", and "mpi_wait", took 31,635 profile samples. Our implementation took just 1,767 profile samples, a speedup of 18×. For the entire run, "global_scatter" went from 4.2% of the runtime to just 0.3%.

The co-array/pointer version was also much simpler to analyze. Consider the following performance report.

```
pat_report -b function,callers
```

This report is useful for determining not only which procedures take the most runtime, but also which calls to each procedure take the most runtime. Procedures internal to the MPI library tend to clog up such profiles because each generates a new entry with a full call stack. For example, here are the "global_scatter" entries pulled out of the report for the MPI version using "grep", with some content on each line removed to conserve space.

```
||3.4%| 3.4% |26048|global_scatter
|||0.5%|14.0% | 3502|global_scatter
|||0.0%|14.9% |    1|global_scatter
|||0.3%|47.9% | 2015|global_scatter
|||0.0%|48.1% |    1|global_scatter
|||0.0%|87.7% |    3|global_scatter
|||0.0%|93.7% |  15|global_scatter
|||||    |      |     |global_scatter
|||||    |      |     |global_scatter
|||0.0%|94.1% |    6|global_scatter
|||||    |      |     |global_scatter
|||||    |      |     |global_scatter
|||0.0%|97.2% |    2|global_scatter
||0.0%|97.6% |    3|global_scatter
||||    |      |     |global_scatter
||||    |      |     |global_scatter
|||    |      |     |global_scatter
|||    |      |     |global_scatter
|||0.0%|     98.7% |    1|global_scatter
|||0.0%|     99.5% |    1|global_scatter
|0.0%|     99.9% |   37|global_scatter
```

The co-array version makes no library calls, so it only appears once in the profile.

```
|0.3%|93.2%| 1767|global_scatter
```

Next consider another flavor of performance report.

```
pat_report -b functions,lines
```

This report shows which lines in each procedure represent a significant amount of runtime. For the MPI version, this report is not useful because all the time is in the MPI calls; none of the lines of "global_scatter" show significant usage. The report for the co-array version is useful, however.

```
|0.3% |93.2% |   1767 |global_scatter
||-----------------------------------
||0.2%|93.2%|1552|line.253
||0.0%|93.2%| 100|line.256
||0.0%|93.2%|  67|line.255
||0.0%|93.2%|  34|line.264
||0.0%|93.2%|  10|line.243
||0.0%|93.2%|   3|line.216
||0.0%|93.2%|   1|line.295
||===================================
```

Most of the time in "global_scatter" was spent on line 253, which is the receiver synchronization loop; the time for memory operations was insignificant. This indicates that synchronization latency or a small load imbalance is the primary source of communication cost.

The dramatic reduction of communication cost and improved ability to analyze performance may make the use of co-arrays and pointers well worth the effort. Expected improvements to the performance of MPI collective operations could reduce the need, however, and improvements to "pat_report" could eliminate the analysis advantage. One should also note that the co-array/pointer technique is not at all portable, and it is prone to subtle errors that can lead to deadlock.

Even with improvements to MPI, direct memory access will likely have significant performance advantages for irregular, latency-bound communication. Future revisions to the CAF definition may need to add functionality like the pointer "trick" described here to make it portable.

References

[1] L. Oliker, R. Biswas, J. Borrill, A. Canning, J. Carter, M.J. Djomehri, H. Shan, and D. Skin-

ner. "A Performance Evaluation of the Cray X1 for Scientific Applications". *VECPAR'04: 6th International Meeting on High Performance Computing for Computational Science*, 2004, to appear.

[2] J.D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. `http://www.cs.virginia.edu/stream/`

[3] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson. "An extended set of FORTRAN Basic Linear Algebra Subprograms". *ACM Trans. Math. Soft.*, 14 (1988), pp. 1–17.

[4] T. Maier and J.B. White III. "Towards Full Simulation of High-Temperature Superconductors". *CUG 2004*.

[5] D. Bailey, T. Harris, W. Sahpir, R. van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0*. Report NAS-95-020, Dec. 1995.

[6] *POP Home*. `http://climate.lanl.gov/Models/POP/`

[7] *MPI—The Message Passing Interface Standard*. `http://www-unix.mcs.anl.gov/mpi/`

[8] *Co-Array Fortran*. `http://www.co-array.org/`

[9] *CICE Home*. `http://climate.lanl.gov/Models/CICE/`